# TO WRITE A PROGRAM = TO FORMULATE ACCURACY

**Jan KASPAR**
Charles University in Prague, Faculty of Mathematics and Physics
Sokolovska 83, 186 00 Praha 8,Czech republic
E-mail: kaspar@karlin.mff.cuni.cz

## ABSTRACT

Mathematics required completely accurate formulation. The control of accuracy could, however, be difficult. The control by our own thinking could be inadequate because even when the formulation is not sufficiently correct, we know what we meant to say and forgive ourselves, the inaccuracy we commit. One possibility, how to prevent this is to control the formulations by a program. Thus will be delegating the checking of our expressions to a computational technology, which will curry it out accurately. In my contribution I will therefor introduce several simple programs and demonstrate how to use them for checking accurate formulations of some problems - in particular solutions of examples from secondary-school level Math. To demonstrate this idea I have chosen a graphing calculator (esp. TI-83), which offers an easily managed, simple, easily understood programming language, which fulfil the requirements of structural programming. At the same time its' capability is sufficient to enable the solutions of practically all problems dealt with in secondary-school Math. The contribution will include examples of algorithms for simple tasks, as well as examples of more complicated problems, where the solution requires an accurate construction of algorithms for partial tasks. The issue of dealing with the verification of accurate formulation is an integral part of the subject "Computational Technology for Teachers of Math", which is included in the study program for students, prospective teachers of Math in the Faculty of Math and Physics at the Charles University in Prague, Czech Rep. This subject was included in study program about 5 years ago as part of modernization of this program with respect to increasing use of Computational Technology in the teaching of Math at the secondary school. In addition the contribution also demonstrate non-standard application of the graphing calculators in teaching Math.

**KEYWORDS**: Algorithm, programming, program, subprogram, accurate formulation, control of accuracy, secondary-school mathematics, graphing calculator

## 1. Introduction

In teaching mathematics we often find that although the students can solve mathematical problems, they cannot adequately describe their solution process step-by-step. Mathematics requires completely accurate formulation. The control of accuracy could, however, be difficult. The control by students' own thinking could be not adequate because even when the formulation is not sufficiently correct, they know what they meant to say and forgive them the inaccuracy they commit. One possibility, how to prevent this is to control the formulations by program. Thus will be delegate the checking accurate formulations of some problems – in particular solutions of examples from secondary-school level Mathematics. To demonstrate this idea I have chosen a graphing calculator (esp. TI-83), which offers an easily managed, simple, easily understood programming language, which fulfils the requirements of structural programming. At the same time its' capability is sufficient to enable the solutions of practically all problems dealt with in secondary-school Mathematics. The contribution includes examples of algorithms for very simple tasks, as well as examples of more complicated problems, where the solution requires an accurate construction of algorithms for partial tasks. In addition the contribution also demonstrate non-standard application of the graphing calculators in teaching Mathematics.

For those not familiar with the GC TI-83, let me just mention that its programming language has, in addition to the usual "input-output" procedures, the following basic commands (among others):

- the conditional test "If Then Else"
- the incrementing loop "For"
- the conditional loops "While" and "Repeat"
- the end of a block signification "End"
- a program as a subroutine execution "prgm"

(and only "one-letter" identifiers - name of variables, are available).

## 2. Examples

Four very simple examples – programs – are introduced for the purpose of demonstration. For each of them detailed commentary is given concerning those commands, where accurate formulation in the description of the relevant calculation algorithm is essential. Such accurate commands are necessary preconditions for the generated program to perform the calculation of a given problem correctly.

**Example 1:**

Write a program for solution of equation $a*x^2 + b*x + c = 0$.

Let me offer you my solution with comments:

```
PROGRAM:QE
:Real :ClrHome          These 3 commands illustrate, that we'll work in real mode, start
:Promt A,B,C            with clear display, and 3 values of coefficients a,b,c are asked
                        (their values are necessary, when we start the solution process)
:Fix 2                  The command for results' edition (2 decimal)
:If  abs(A) < 10^(-6)   Very important part of program. We have to discuss all
:Then                   possibilities of the coefficients (a,b,c) values and have to finish
:If  abs(B) < 10^(-6)   solution in each of branch.
:Then                   As a "side effect" in this discussion is test, if a value of each of
```

```
:If  abs(C) < 10^(-6)          coefficients (a,b,c) = 0. Because they are stored as a real
:Then                          variables in calculator, it is not good idea to use "direct" test
:Output(5,1,"ALL X")           (If "variable = 0"). For real variables is better to test
:Else                          "if variable ∈ or ∉ of ε- neighbourhood of 0".
:Output(5,1,"NO X")            This part of program also shows, how to include If ... Then ...
:End                           Else ... End command into the other If ... Then ... Else ... End
:Else                          command
:-C/B–>X  :Disp X
:End
:Else
:B^2–4*A*C–>D
:If D<0                        We must respect then if D<0 the complex mode is necessary.
:Then :a+bi :End               And there is shown If ... Then ... End command (without Else)
:(-B–√(D))/2/A–>X              Very important command in this form; it illustrates that equal
                               priority of operations must be respect.
:(-B+√(D))/(2*A)–>Y            And in this command there is illustrated how the brackets
                               influate priority of operations and help us to use for
                               formulas "pretty" notation.
:Disp X :Disp Y :Stop
```

**Example 2.**

There is shown, in two following examples, how it is useful to choose "more sufficient" of two conditional loops with different philosophy:
**"while" loop** – at first the condition is tested; if "true", all commands inside loop are executed and the loop is repeated, if "false", no command inside loop is executed and the following command after "while" loop is executed
**"repeat" loop** – commands inside loop are executed and when the execution of these commands is finished, the condition is tested; if "true", "loop" is finished and the following command after "repeat" loop is executed, if "false", loop is repeated.

This moment is very important indicator, if we understand what we wish to do, what we wish to achieve.

**Problem 2.1** Two integers *A* and *B* are given. Use the Euclid algorithm to evaluate the greatest common divisor.

The program is very simple, of course:

```
PROGRAM: GCD
:Prompt A, B
:While A≠B                     In this case it is better to use the "while" loop, because if given
:If A>B                        values of A and B are equal, we do not have to do anything, "while"
:Then                          loop gives us the result directly. On the other hand it would be
:A–B–>A                        difficult to formulate the condition for the "repeat" loop.
:Else
:B–A–>B
:End
:End
:Disp A:Stop
```

Sometimes, if "repeat" loop is used when it is not suitable, one "back-step" is necessary,.

**Problem 2.2** Two integers *A* and *B* are given. Compute the *A : B* ( = quotient *Q* with reminder *R*, *R*∈⟨0,*A*) ), using only operation + and – and tests.

If we use "while" loop, all is without problem:

```
PROGRAM DIV1
:Prompt A,B
:0->Q
:While A≥B
:A-B->A                          I hope, all is clear and easy without any comment
:Q+1->Q
:End
:A->R
:Disp Q
:Disp R
:Stop
```

If we use "repeat" loop, there are two different situations:

```
PROGRAM DIV2
:Prompt A,B
:0->Q
:A->R
:Repeat R<B
:R-B->R
:Q+1->Q
:End
:Disp Q
:Disp R
:Stop
```

For 13 : 5 e.g. is all OK, the results in individual steps of the loop are:
1) $R$ = 13 – 5 ( = 8), $Q$ = 1, $R < B$ (8 < 5) is "false" (test is the last provided command in "repeat" loop !)
2) $R$ = 8 – 5 ( = 3), $Q$ = 2, $R < B$ (3 < 5) is "true", "repeat" loop is finished in this moment
Command :Disp *Q* gives correct quotient = 2 and in *R* is correct reminder 3.

But if we use this program for 2 : 5, the results in individual steps of the loop are:
$R$ = 2 – 5 (= -3), $Q$ = 1, $R < B$ ( -3 < 5) is "true" , "repeat" loop is finished, but the result is not convenient (reminder *R* isn't from required interval ⟨0,*A*) and *Q*=1 is wrong quotient). In this moment it is necessary to do "back-step", mentioned before, i.e. we have to repair it by following commands after :End of "repeat" loop:

```
:If
:R< 0
```

```
:Then
:Q-1->Q
:R+B->R
…
```

But we know only just now, why this "artificial" group of commands was included into the program. Be sure, it will be forgotten during very short time, and for the "second" person, who will use this our program, it is only as a puzzle! Similar situation is very helpful indicator, that "while" loop was better for this problem.

In this moment also let me mention, that if we need the values of variables *A* and *B* saved in original (e.g. for the output: "The GCD of A and B is …" ), it is necessary to move them into two auxiliary variables and all commands provide with these auxiliary variables. Their values are changed during calculation, but the values of variables *A* and *B* are saved. By the way it is illustrated in Problem 2.2. Value of variable *A* there is moved into variable *R*, its' values are changed and *A* has original value when work of program is finished.

For "Repeat" loop let me offer the next example:

**Problem 2.3:** Evaluate the sum of the series $\Sigma\ (1/2)^n$ ($n = 0, 1 \ldots$ ) with given accuracy *e*.

The program is very easy, again:

```
PROGRAM: SUM
:Prompt E
:0–>S :0–>N  :1–>D
:Repeat D≤E
:N+1–>N
:(1/2)^N–>D
:S+D–>S
:End
:Disp S:Stop
```

*It's clear in this example, that we have to go through the block of commands inside "repeat" loop at least once, to obtain required accuracy.*

There is another very important moment. It isn't necessary to evaluate in each step (1/2)^2, (1/2)^3 etc. When we are evaluating $(1/2)^i$ it is better to compute it as $(1/2)^{i-1} * (1/2)$; in the program we cancel the commands :0–>N and :N+1–>N and the 5th line we substitute by command :1/2∗D–>D)

In the last example I would like to illustrate, how it is helpful for description (and understanding) of solution of large, complicated problem, if that solution is split into several solutions of partial, simple problems.

**Example 3:**

The line *p* and two different points *A, B* in the same half-plane are given, non of which lies on the line *p*. Find such a point *P* on the line *p*, that the sum of lengths of segments *AP* and *BP* is minimal.

**Solution.**

What we have to do, when we solve this example "with pencil and paper":

- we have to find an image point *A*' of the point *A* in symmetry with respect the line *p*
- we have to draw line *q*, passing though points *A*' and *B*

- we have to find the result point *P* as an intersect of lines *p* and *q*

But the first step isn't elementary, there are three steps inside, in fact:

- we have to draw perpendicular line *k* to line *p* passing through point *A*
- we have to find intersection of lines *p* and *k* – point *Q*
- we have to find point *A'* on line *k*, in opposite half-plane than point *A* is, to be *AQ* and *A'Q* equal segments

Now, when the example is analysed, we can start to write program or subprograms for each elementary step. Because we solve this example in plane, a point is described by two coordinates *x* and *y* and a line is described by general equation $a*x+b*y+c=0$. There is very easy solution for this (and similar) situation:

Let me suppose that we have written the following subprograms for partial, simple calculation:
prgm PLP – subprogram returns three coefficients of equation of line, perpendicular to given line (by three coef. of eq.) and passing through given point (by two coord.)
prgm PPL – subprogram returns three coefficients of equation of line, passing through two given points (by two coord.)
prgm LLP – subprogram returns two coordinates of point, intersection of two given lines (by three coef. of eq.)

and two subprograms for "input point" and "input line":

prgm RP – input of point (two coordinates, x and y)
prgm RL – input of line (three coefficients of general equation, *a, b, c*).

The last subprogram for the construction of the point *A'*, symmetric point with given point *A* with respect to the line *p*, is missing. This problem we can solve, using vectors (vector *AA'* = 2* vector *AQ*). It's useful to write subprogram not for "2*vector" but for "*n**vector" and use *n* = 2. Let us assume that we have written such a subprogram

prgm NV – subprogram returns two coordinates of end point *E* of vector *AE* = *n**vector*AB* where *A* is starting point of both vectors and *B* is end point of given vector

Sometimes the students are very surprised, why there are written two subprograms for "input point" and for "input line" separately and why the operation "input point" and "input line" are not included into subprograms LPP, RP and RL. The answer is very easy (and reason is very important and strong!) – sometimes we use for these subprograms points or lines, these are the results of previous calculation. In that moment these values are automatically disposable and on the other hand, it may be very inconveniently to have to input these values, in fact, once more.

And now we can write very easy final program. It is sequence of subprograms:

```
… :prgm RP
```

```
:prgm RP
:prgm RL
:prgm PLP              In fact, the solution of "large, complicated" problem there is
:prgm LLP              described very precisely by solution of partial, simple problems
:prgm NV   (for n=2)
:prgm PPL
:prgm PLP …
```

And having these subprograms (solution of partial, simple problems), we can describe precisely solution of a lot of other problems (triangle's midpoint or centroid, etc., etc).

## 3. Short resume

An important indication of the need to improve the description of a mathematical procedure is to find that the program leads to incorrect results. In such a case it is quite likely that step-by-step description used in programming has not been correct.